On Finding Frequent Patterns in Directed Acyclic Graphs*

Andrea Campagna
IT University of Copenhagen
DK-2300 København S
Denmark
acam@itu.dk

Rasmus Pagh
IT University of Copenhagen
DK-2300 København S
Denmark
pagh@itu.dk

ABSTRACT

Given a directed acyclic graph with labeled vertices, we consider the problem of finding the most common label sequences ("traces") among all paths in the graph (of some maximum length m). Since the number of paths can be huge, we propose novel algorithms whose time complexity depends only on the size of the graph, and on the relative frequency ε of the most frequent traces. In addition, we apply techniques from streaming algorithms to achieve space usage that depends only on ε, and not on the number of distinct traces.

The abstract problem considered models a variety of tasks concerning finding frequent patterns in event sequences. Our motivation comes from working with a data set of 2 million RFID readings from baggage trolleys at Copenhagen Airport. The question of finding frequent passenger movement patterns is mapped to the above problem. We report on experimental findings for this data set.

1. INTRODUCTION

Sequential pattern mining has attracted a lot of interest in recent years. However, some of the probabilistic techniques that have proven their efficiency in mining of frequent itemsets have, to our best knowledge, not been transferred to the realm of sequence mining. The aim of this paper is to take a step in that direction, namely, we propose an analogue of Toivonen's sampling-based algorithm for frequent itemset mining [15] in the context of sequential patterns.

At a conceptual level we work with a new, simple formulation of the problem: The input is a directed acyclic graph (DAG) where the vertices are events and there is an edge between two events if they are considered to be connected (i.e., part of the same event sequences). Vertices are labeled by the type of event they represent. This allows certain flexibility in modeling that is lacking in many other formulations:

- Spatio-temporal events can be connected based on both spatial and temporal closeness.
- Events that have an associated time range (rather than a single time stamp) can be connected based on an arbitrary closeness criterion.

The data mining task we consider is to find the most common sequences of event types ("traces") among all paths in the DAG, or more generally all paths of some maximum length m. The challenge is to handle the huge number of paths that may be present in a DAG. Our approach rests on a novel sampling procedure that is able to create a sample of any desired size, in time that is linear in the size of the DAG (for preprocessing) and the size of the sample (for sampling). This allows a time complexity for the mining procedure that depends on the relative frequency ε of the most common traces rather than the total number of traces. We also apply a technique from data streaming algorithms to achieve space that depends on ε rather than on the number of distinct traces.

Though our formulation does not capture all the many aspects present in other approaches to sequential pattern mining, we believe that it possesses an attractive combination of *expressive modeling* and *algorithmic tractability*.

1.1 Problem definition

We are given a directed acyclic graph $G = (V, E)$, and a function label: $V \to L$ that maps vertices to their labels. A path p in G is a sequence of vertices $v_1, v_2, \ldots, v_j \in V$ such that $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, j-1$. A path p has a *trace* label(p), which is the vector of labels on the path. Let S_m denote the multiset of all path traces of length at most m, i.e.,

$$S_m = \{\text{label}(p) \mid p \text{ is a path in } G \text{ of length at most } m\} \ .$$

The data mining task is to find the most frequent traces in S_m. It comes in several flavors:

- **Top-k**. For a parameter k, find the k traces that have the most occurrences in S_m (breaking ties arbitrarily).

*This work was supported in part by the SPOPOS project, supported by the Danish Research and Innovation Agency under the Danish Ministry for Knowledge, Technology and Development.

- **Frequency** ε. Find the set of traces that have relative frequency ε or more in S_m.
- **Monte Carlo**. For both the above variants we can allow an error probability δ (typically allowing a false negative probability, i.e., that we fail to report a trace with probability δ).

In this paper emphasis will be on Monte Carlo algorithms for the frequency variant. However, we note one can also obtain results for top-k by a simple reduction.

1.2 Related work

There is a large body of related work on sequence data mining, see e.g. [12, 14, 8, 6, 18, 5, 2, 13]. These works deviate from the present one in that they consider the input as a sequence of timestamped events, and allow a host of formulations of what kinds of subsequences are of interest. In contrast, we put the modeling of interesting subsequences into the description of the event sequence (by defining DAG edges), and the patterns sought are simple strings. This allows us to do things that we believe have not been done, and are probably difficult, in traditional sequential data mining settings, namely making use of sampling methods. The difficulty with sampling is, of course, that patterns can overlap in complicated ways, so any straightforward approach (such as sampling nodes or edges) will fail to give independent samples.

Another related area is algorithms for finding frequent subgraphs in graphs, see e.g. [17, 7, 11, 4]. Indeed, the problem we consider can be seen as that of finding frequent (labeled) paths in an acyclic graph. Our work deviates from previous works mainly in that we consider directed acyclic graphs rather than general (undirected) graphs. This allows us to present algorithms with provable upper bounds on space usage and running time. No such efficient bounds are possible for general graphs: Even the problem of determining if a graph contains a simple path of length k requires time exponential in k [1, 16], and this is inevitable assuming the hamilton cycle problem requires exponential time in the number of vertices (a well-established hypothesis). In addition, we believe that this is the first use of sampling methods in the context of finding frequent subgraphs. Possibly, this could inspire further work on using sampling in graph mining.

2. OUR SOLUTION

2.1 Generation of all traces

As a warmup we consider the task of producing the multiset of all traces having maximum length m. We will use the notation $S_i(v)$ to denote the multiset of traces corresponding to paths (of length at most m) starting in node v. Clearly $S_0(v) = \emptyset$. For $i > 0$ we have the recursive definition

$$S_i(v) = \{\text{label}(v)\} \times (\epsilon \cup \bigcup_{v',(v,v')\in E} S_{i-1}(v')),$$

where ϵ denotes the empty trace, and $\bigcup$ is multiset union. Clearly we have $S_m = \bigcup_{v\in V} S_m(v)$.

These equalities lead to a simple recursive algorithm, shown in Figure 1. It is easy to see that if traces are represented in a reasonable way (e.g. as singly linked lists) the running time is linear in the size $|V|+|E|$ of the graph and the total length of the traces generated.

```
 1: procedure AllTraces(v, t, i)
 2:     if i > 0 then
 3:         output t||label(v)
 4:         for each v' where (v, v') ∈ E do
 5:             AllTraces(v', t||label(v), i − 1)
 6:         end for
 7:     end if
 8: end procedure
 9: for v ∈ V do
10:     AllTraces(v, ϵ, m)
11: end for
```

Figure 1: The procedure AllTraces outputs the concatenation of a trace prefix t, and each trace starting at v having length at most i. The notation $||$ is for concatenation of traces. Lines 7–9 call AllTraces for all vertices v, with the empty trace ϵ as prefix, producing the multiset S_m of all traces of length at most m.

Succinct output. If we are satisfied with returning hash values of the traces (unique with high probability) the time can be improved such that only $\mathcal{O}(1)$ time is used for each trace, i.e. time $\mathcal{O}(|V|+|E|+|S_m|)$ in total. This can be done using a standard incremental string hashing method such as Karp-Rabin [9]. Observe that the output is sufficient to find the *hash values* of the most frequent traces in S_m (with a negligible error probability). A second run of the procedure could then output the actual frequent traces, e.g. by looking up the count of each hash value computed.

2.2 Generation of a random sample

If the patterns we are interested in occur many times, substantial savings in time can be obtained by employing a sampling procedure. That is, rather than generating S_m explicitly we are interested in an algorithm that produces each trace in S_m with a given probability p, independently. This will reduce the expected number of samples to a fraction p of the original. The choice of p is constrained by the fact that we still want to sample each frequent trace a fair number of times (to minimize the probability of *false negatives* being introduced by the sampling).

Counting phase. Our algorithm starts by computing, for $i = 1, \ldots, m$ the number of paths $v.c[i]$ of length at most i that start in each vertex v. We assume that this can be done using standard precision (e.g. 64 bit) integers. The algorithm shown in Figure 2 mimics the structure of the naïve generation algorithm, but uses memoization (aka. dynamic programming) to reduce the running time.

For each $i \leq m$ the cost of all calls to CountTraces with parameters (v, i), disregarding the cost of recursive calls, is easily seen to be proportional to the number of edges incident to v. This means that the total time complexity of the counting phase is $\mathcal{O}(|E|m)$. The space usage is dominated by an array of size m for each vertex, i.e., it is $\mathcal{O}(|V|m)$.

```
1: function CountTraces(v, i)
2:     if v.c[i] =null then
3:         v.c[i] ← 1
4:         for each v' where (v, v') ∈ E do
5:             v.c[i] ← v.c[i]+CountTraces(v', i − 1)
6:         end for
7:     end if
8:     return v.c[i]
9: end function

10: for v ∈ V do
11:     CountTraces(v, m)
12: end for
```

Figure 2: Recursive computation of the paths of traces for each starting vertex, using memoization. It assumes that each value $v.c[0]$ is initially set to zero, and each value $v.c[i]$, $0 < i \leq m$, is initially null.

Sampling phase. Consider the multiset $S_i(v)$ of traces, which has size $v.c[i]$ by definition. The probability that none of these traces are sampled should be $(1-p)^{v.c[i]}$. Conditioned on the event that at least one trace from $S_i(v)$ is sampled, we either have to sample a trace of length more than one (starting with label(v)), or include the trace $\{v\}$ in the sample. In a nutshell, this is what the procedure SampleTraces of Figure 3 does.

Let rand() denote a function the returns a uniformly random number in $[0;1]$, independently of previously returned values. The condition rand() $> (1-p)^{v.c[m]}$ holds with probability $1-(1-p)^{v.c[m]}$, so lines 14–16 call SampleTraces if and only if we need to sample at least one trace from $S_m(v)$. In the procedure SampleTraces we use, similarly to above, a parameter t to pass along a trace prefix. The variable *out* is used to keep track of whether a trace has been output in the recursive calls. If *out* is false after all recursive calls we sample $t||\text{label}(v)$. For each v' with $(v,v') \in E$ the probability that we do *not* sample any trace from $\text{label}(v)||S_{i-1}(v')$ is $(1-p)^{v'.c[i-1]}/(1-(1-p)^{v.c[i]})$. This is exactly the correct probability since we condition on at least one trace in $S_i(v)$ being sampled.

Refinement. Observe that the probability in line 4 may be precomputed for each edge and value of i. Even with this optimization, a direct implementation of the pseudocode in Figure 3 may spend a lot of time in the **for** loop of SampleTraces without producing any output. To get a theoretically satisfying solution we may preprocess, for each (v,i), the probabilities $p_1, p_2, \ldots, p_d$ of making the recursive calls. Specifically, for $j = 0, \ldots, d$ we consider the probabilities $q_j = \Pi_{j' \leq j}(1-p_{j'})$ that no recursive call is made in the first j iterations. If we choose r uniformly at random in $[0;1]$ then the probability that $q_{j-1} > r > q_j$ is exactly the probability that the first recursive call is in the jth iteration. Similarly, the probability that $r > q_d$ is exactly the probability that no recursive call is made. Thus, by doing a binary search for r over $q_d, \ldots, q_0$ we may choose, with the correct probability, the first iteration j_1 in which there should be a recursive call. The same method can be repeated, using a random value r in $[0; q_{j_1}]$ to find the next recursive call, and so on.

```
1: procedure SampleTraces(v, t, i)
2:     out ← false
3:     for each v' where (v, v') ∈ E do
4:         if rand()> (1 − p)^{v'.c[i−1]}/(1 − (1 − p)^{v.c[i]}) then
5:             SampleTraces(v', t||label(v), i − 1)
6:             out ← true
7:         end if
8:     end for
9:     if out = false or rand()< p then
10:        output t||label(v)
11:    end if
12: end procedure

13: for v ∈ V do
14:    if rand()> (1 − p)^{v.c[m]} then
15:        SampleTraces(v, ϵ, m)
16:    end if
17: end for
```

Figure 3: The procedure SampleTraces outputs the concatenation of a trace prefix t and a random sample of the traces starting at v of length at most i. The traces are sampled from the conditional distribution that is guaranteed to sample at least one trace. As before, the notation $||$ is for concatenation of traces, and ϵ denotes the empty trace. Lines 13–17 call SampleTraces for each vertex v with probability $1-(1-p)^{v.c[i]}$, to produce a sample of all traces starting at v having length at most i, where each trace is chosen independently at random with probability p.

In the worst case this uses time $\mathcal{O}(\log |V|)$ per recursive call. We can exploit the fact that we are searching for a random value r to decrease this to $\mathcal{O}(1)$ expected time. The basic idea is to place the probabilities q_j in buckets according to the $\log d$ most significant bits, and furthermore store in each bucket its predecessor (i.e., the maximum j such that q_j is smaller than all elements in the bucket). Given r, we can find j_1 by inspecting the values in the bucket that r belongs to (the elements, and their predecessor). This will take expected time $\mathcal{O}(1)$ since r is random and the average number of values per bucket is 1.

To make this work not just for the first search, we adjust the bucketing as follows: We partition $q_1, \ldots, q_d$ according to the number of leading 0s in the binary representations (wlog. there are $\mathcal{O}(\log n)$, since we can rely on brute-force search for low probability events, i.e., if r gets very small). In each partition, containing d' values, we partition the values in buckets according to the $\log d'$ most significant bits. As before, we store the predecessor of each bucket. It is clear that this data structure requires $\mathcal{O}(d)$ space, and can be constructed in time $\mathcal{O}(d)$. A search for random r in $[0;\gamma]$ happens in the structure corresponding to the number of leading 0s in γ. This will choose a random bucket of expected size $\mathcal{O}(1)$, and the analysis finishes as before. If there are no q_j values with the right number of leading 0s, we use a special structure of $\mathcal{O}(\log n)$ bits to find the partition of the predecessor in $\mathcal{O}(1)$ time.

As before, we can choose to have a succinct output where traces are represented by the hash values of their traces, with no increase in time complexity.

2.3 Time and error analysis

For the time analysis we focus on the refined implementation described above, since it allows a clean and exact theoretical analysis. A similar analysis of the version stated in the pseudocode can be made under the assumption that the outdegree of vertices in G is bounded by a constant. Observe that if SampleTraces makes c recursive calls this takes expected time $O(1+c)$. Also observe that the total number of procedure calls is upper bounded by the total length of all sampled traces — this is because each recursive call is guaranteed to produce at least one output. Combining these facts we see that the expected time for all calls to SampleTraces is linear in the length ℓ of all traces sampled. Notice that the expected value of ℓ is $\mathcal{O}(p|S_m|m)$. Since ℓ is independent of the random choices determining the running time of the data structure in the refined implementation we can conclude that the total expected running time of the code in Figures 2 and 3 is $\mathcal{O}(|V| + |E|m + p|S_m|m)$.

The parameter p must be chosen such that $p = C/\varepsilon$, where $C > 1$ is a parameter that determines the false negative probability. The expected number of times that we sample a trace with frequency ε' is $C\varepsilon'/\varepsilon$, and since the samples are independent the number of samples follows a binomial distribution. By Chernoff bounds, this means that if $\varepsilon' \geq \varepsilon$ then the number of samples is at least $C/2$ with probability $1 - 2^{-\Omega(C)}$. Concrete error probabilities for $C = 10$ are discussed in our experimental section. We have the following theoretical result:

Theorem 1. *We can generate a random sample of S_m in expected time $\mathcal{O}(|V| + |E|m + \log(1/\delta)/\varepsilon)$ such that each trace with frequency ε or more has frequency at least $\varepsilon/2$ in the random sample with probability $1 - \delta$.*

Observe that the running time is independent of the total number of traces in S_m.

2.4 Putting things together

It remains to assess how to choose, among the samples, the ones that are actually interesting. In particular, we are interested in those traces appearing in the sample at least $C/2$ times.

This problem can be efficiently addressed used a *frequent items* algorithm. Such algorithms have been designed for use in a data streaming context, and guarantee low space usage. A comprehensive treatment and an experimental comparison between various techniques can be found in [3]. The problem itself dates back at least to the 1980s, and can be formalized in this way:

Definition 2. *Given a stream $\mathcal{S}$ of n elements and a frequency threshold η, the frequent items problem asks for the set $\mathcal{F}$ of items that occur at least η times.*

The algorithms addressing this problem usually solve a relaxed version where a modest number of false positives can appear in the output, since this reduces the space requirements to $\mathcal{O}(n/\eta)$. For completeness, we describe a concrete frequent items implementation in Appendix A.

In order to solve the frequent items problem without false positives, which in our case means without reporting traces whose frequency is below ε, we will make two passes, i.e., generate the sample twice and do exact counting of potentially frequent items in the second pass. This will roughly double the running time.

Lemma 3. *Given a stream of elements representing the set of samples of traces produced by SampleTraces, the space needed in order to output the traces with frequency at least ε, without producing any trace with frequency less than ε, is $\mathcal{O}(1/\varepsilon)$ words.*

Let $\text{freq}(t, S_m)$ denote t's fraction of S_m (viewed as a multiset). E.g., if $S_2 = \{aa, aa, ab, ba, bb\}$ we have $\text{freq}(aa, S_2) = 2/5$. Putting together Theorem 1 and the above lemma, we get:

Theorem 4. *Let ε and δ be positive reals. In expected time $\mathcal{O}(|V| + |E|m + \log(1/\delta)/\varepsilon)$ and space $\mathcal{O}(1/\varepsilon)$ we can produce a set T of $\mathcal{O}(1/\varepsilon)$ traces, and accompanying random variables X_t, $t \in T$, such that:*

- *For each t with $\text{freq}(t, S_m) \geq \varepsilon$, $\Pr[t \in T] \geq 1 - \delta$, and*
- *for each $t \in S$, X_t has binomial distribution with mean $\text{freq}(t, S_m) f(\varepsilon, \delta)$, where $f(\varepsilon, \delta) = \Theta(\log(1/\delta)/\varepsilon)$.*

The first property says that the probability that a frequent trace is not reported is at most δ. The second property says that the frequency of the traces in T can be estimated, with strong statistical guarantees, since the X_t values come from a highly concentrated distribution with mean proportional to $\text{freq}(t, S_m)$.

3. FROM EVENT SEQUENCE TO A DAG

An event sequence is a set S of tuples of the form (t, i, ℓ), where $t \in \mathbb{R}$ is a time stamp, i is a tag identifier, and ℓ is a label (in our case, ℓ is a location identifier that indicates an approximate location, namely vicinity of an antenna). In this work we do not consider the physical locations of antenna as part of the input.

Formally we may define the problem as follows: For a given number Δ, the input set specifies a directed acyclic graph $G_\Delta = (V, E_\Delta)$, where each observation is a vertex, and there is an edge from v_1 to v_2 if and only if the vertices are observations of the same tag, at different locations, separated by at most Δ time units (we use minutes as the time unit from now on).

To produce the DAG we sort the data by tag ID and timestamp. Note that this makes it easy to find all the edges from a particular vertex v in G_Δ: Simply scan the sorted list forward until either the timestamp differs by more than Δ from that of v, or we reach a node corresponding to another tag.

Example. If $\Delta = 20$ and we observe locations 1, 2, 3, 6, 7 at time 10, 20, 30, 60, 70, the following subsequences are

considered to reflect a movement: 1-2, 2-3, 1-2-3, 1-3, 6-7. Notice the inclusion of 1-3, where one observation is skipped, since there is at most Δ minutes between the observation of 1 and 3.

3.1 Converting the RFID data

We have worked with a data set consisting of readings of RFID (Radio-Frequency ID) tags by fixed-position antenna. RFID chips can be identified only when they are in the proximity of an antenna, which means that readings give approximate information about the location of an RFID tag. Such data sets, as well as similar data sets based on other technologies, are becoming increasingly available as more and more items, from parcels to items in shops, are being tagged with RFID chips.

Before using the RFID data to create a DAG, we have cleaned some of the noise present in the data. One source of noise was the presence of sequences of readings regarding trolleys remaining in zones where the range of two antennas is overlapping. This gave rise to sequences of alternating readings of the form $(x^+y^+)(x^+y^+)^+$. In order to clean up these interferences, we replaced such sequences by a new zone label that represents the zone of overlap of the range of antennas. In particular we have used, for a sequence $(x^+y^+)(x^+y^+)^+$, the label $\min\{x,y\}*100+\max\{x,y\}$. This can be thought of as an increase in the spatial resolution of the readings.

Another source of noise, sometimes connected with the one just described, is the presence of sequences of readings regarding the same zone for a given trolley. In order to avoid having traces of the form $t = (Vyy^+W)$, where V and W are sequences of readings, we considered only one occurrence of y, properly managing the timestamps of the readings. In particular this means that, assuming the difference in time between any two consecutive y is within the threshold Δ, in the DAG we put a directed edge (v, y), $v \in V$ iff the first occurrence of y after V occurred within time Δ from v. Moreover we put a directed edge (y, w), $w \in W$ iff w happened within time Δ from the last reading of y in t.

4. EXPERIMENTS

For the experiments we have used the RFID dataset described above. We have used this dataset since it suits quite well the needs of the abstract formulation of the problem, and is massive enough to be challenging for our algorithm. Moreover, did not manage to find interesting, raw DAG data. However, it would be of interest to try our algorithms on DAGs derived from other (publicly available) data sets.

We ran a set of experiments on the data, in order to understand how many patterns would have been generated for a given Δ and a size m. Fig. 6 shows the size of the graph for different sizes of Δ. We compared the obtained results with the expected performance of our algorithm (from the theoretical analysis). For space usage this gives a rather precise idea about the savings that can be obtained. For time usage, there is greater uncertainty, since the time is influenced by the constant factors in the implementation (which again depends on the hardware on which we run the experiments). It would be of interest to investigate the performance of a concrete, tuned implementation to see how close one can get to the theoretical gains.

Δ	|V|	|E|
20	2206302	4059250
10	2206302	2657931
5	2206302	1721448
3	2206302	1228759

Figure 5: Size of the airport DAG for different values of Δ. As can be seen all graphs are quite sparse, and in fact many nodes have no outgoing edges. This is due to a relatively low resolution in the data set.

Δ	m	Tot. traces	Dis. traces	top 100th	ratio
20	5	365818472	4311942	168000	990
10	5	106678064	1712646	52951	425
10	3	6196850	50085	9458	38.2
5	5	66947355	631300	42008	198
3	5	23152990	280454	15363	93

Figure 6: Characteristics of the data for several combinations of Δ and m. The third column, Tot. traces, represents the total number of traces that would be generated by the naïve approach; the Dis. traces column represents the number of distinc traces; the top 100th column contains the frequency of the 100th most frequent trace; the column ratio represents the saving we would achive using a frequency threshold equal to the one represented in the top 100th column.

Fig.6 reports some interesting characteristics of the data when varying Δ and m. In particular the table contains the number of traces generated, the frequency of the 100th most frequent trace and the ratio between the space needed in case of an exact computation and the space required when our algorithm is used. Note that the space to represent the DAG and the counts is not taken into account in this ratio. The rationale for this is that as we consider longer event sequences the space for the DAG representation is expected to become negligible compared to the space needed for finding the most common traces.

From the results of the test it is clear that great savings can be achieved when the frequencies we are interested in are not too low. In a case, nearly 3 orders of magnitude of space can be saved using our approach.

Fig. 7 shows the number of samples we would take in expectation when $C = 10$ is used. The table gives the flavor of the saving in time that could be achieved with respect to generating all the possible traces. It is worth noticing that with $C = 10$ we would end up with a probability of reporting a false positive that is lower than 7% (this can be seen by considering the probability that a Poisson random variable with mean 10 or more has value less than 5). Here we notice that the total number of traces is already 1–2 orders of magnitude larger than the size of the DAG, so we expect an improvement in running time of at least 1 order of magnitude.

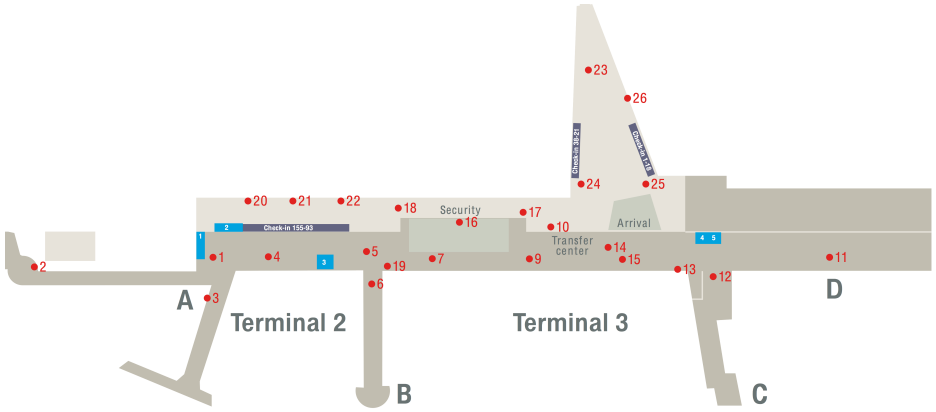

Figure 4: RFID antenna in Copenhagen Airport.

Δ	m	Tot. traces	# samples	ratio
20	5	365818472	22774	16800
10	5	106678064	20147	5295
10	3	6196850	6552	946
5	5	66947355	15937	4200
3	5	23152990	15070	1536

Figure 7: The ratio between the total number of traces and the number of samples we would take using $C = 10$. Whit this value of C, the probability of having false negatives would be approximately 7%

5. REFERENCES

[1] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, July 1995.

[2] Y.-L. Chen and Y.-H. Hu. Constraint-based sequential pattern mining: The consideration of recency and compactness. *Decision Support Systems*, 42(2):1203 – 1215, 2006.

[3] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. *PVLDB*, 1(2):1530–1541, 2008.

[4] M. Fiedler and C. Borgelt. Subgraph support in a single large graph. In *ICDM Workshops*, pages 399–404. IEEE Computer Society, 2007.

[5] F. Giannotti, M. Nanni, D. Pedreschi, and F. Pinelli. Mining sequences with temporal annotations. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 593–597, New York, NY, USA, 2006. ACM.

[6] S. K. Harms, J. S. Deogun, and T. Tadesse. Discovering sequential association rules with constraints and time lags in multiple sequences. In *ISMIS '02: Proceedings of the 13th International Symposium on Foundations of Intelligent Systems*, pages 432–441, London, UK, 2002. Springer-Verlag.

[7] J. Huan, W. W. 0010, J. Prins, and J. Yang. SPIN: mining maximal frequent subgraphs from graph databases. In W. Kim, R. Kohavi, J. Gehrke, and W. DuMouchel, editors, *KDD*, pages 581–586. ACM, 2004.

[8] M. V. Joshi, , G. Karypis, and V. Kumar. A universal formulation of sequential patterns. In *Proceedings of the KDD'2001 workshop on Temporal Data Mining, San Fransisco, August 2001*.

[9] R. M. Karp and M. O. Rabin. Efficient randomized pattern–matching algorithms. *IBM Journal of Research and Development*, 32:249–260, 1987.

[10] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28:51–55, 2003.

[11] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph*. *Data Min. Knowl. Discov*, 11(3):243–271, 2005.

[12] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov*, 1(3):259–289, 1997.

[13] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE '01: Proceedings of the 17th International Conference on Data Engineering*, page 215, Washington, DC, USA, 2001. IEEE Computer Society.

[14] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings*, volume 1057 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 1996.

[15] H. Toivonen. Sampling large databases for association rules. In *Proceedings of the Twenty-Second International Conference on Very Large Data Bases (VLDB '96)*, pages 134–145, San Francisco, Ca., USA, Sept. 1996. Morgan Kaufmann.

[16] R. Williams. Finding paths of length k in $O^*(2^k)$ time. *Inf. Process. Lett*, 109(6):315–318, 2009.

[17] X. Yan and J. Han. CloseGraph: mining closed frequent graph patterns. In P. Domingos, C. Faloutsos, T. SEnator, H. Kargupta, and L. Getoor, editors, *Proceedings of the ninth ACM SIGKDD International Conference on Knowledge*

Discovery and Data Mining (KDD-03), pages 286–295, New York, Aug. 24–27 2003. ACM Press.

[18] Q. Zhao and S. Bhowmick. Sequential pattern mining: a survey. Technical report, School of Computer Engineering, Nanyang Technological University, Singapore, 2003.

APPENDIX

A. A CONCRETE FREQUENT ITEMS IMPLEMENTATION

For completeness, we will describe in a high level fashion one of the several frequent items algorithms existing in literature. The algorithm is presented in [10]. We are interested in reporting the traces appearing at least $C/2$ times in the sample. For this purpose we maintain a set of $2p|S_m|/C$ entries; each entry contains the label of the trace and a counter. Every time SampleTraces outputs a trace t, we look at the set of entries and depending on whether the trace is already recorded in one of the entries or not, we take one of two choices:

- t **appears in entry** i**:** we add 1 the counter associated with the entry i;
- t **does not appear in any entry:** we decrease by 1 all the counters; if a counter reaches 0 we remove the corresponding trace from the entry.

This algorithm guarantees to find all the traces with frequency above the threshold $C/2$, but could return traces with frequency below the threshold. In order to eliminate this traces from the output, a second pass over the sample is required to get exact occurrence counts. There are two possible ways of doing this: Either one can generate exactly the same sample again (using a pseudorandom generator with the same seed, or simply by storing the random choices made). The other way (which is what we analyze theoretically) is to take a new, random sample and count exactly the number of occurrences of those elements that were found to be "possibly frequent" in the first sample. This increases the probability of false negatives by almost a factor of 2, so to compensate for this one needs to slightly increase C.